**Önlabor – SPH folyadék szimuláció (általános számítások a GPUn)**

Horváth Ákos – DKILK6

Gurubi Barnabás - DXEXVR

Konzulens: **Tóth Balázs**

Mit is jelent az a GPGPU?

Alapvetően, ha programozásról beszélünk mindenkinek elsőre az egyszálú (esetleg pár szálú) CPU-n futó programok jutnak az eszébe. Ez miért is jelenthet problémát? Vannak olyan feladatok, amelyek nagymértékben párhuzamosíthatóak, és ha ezt észrevesszük és ki is használjuk sokkal gyorsabb programokat érhetünk el. Mint tudjuk a CPU csak korlátozottan képes a többszálúságra így az olyan problémák esetén, ahol egyszerű számításokat tudunk nagy számban párhuzamosan végezni érdemes a GPU-hoz fordulni. Ilyenkor beszélünk GPGPU alkalmazásról. (GPGPU = General-purpose computing on graphics processing units, azaz általános célú számítások a grafikus processzoron)

Ahhoz persze hogy a GPU-n ilyen dolgokat végezhessünk, természetesen szükség van a megfelelő eszközökre, gondolok itt mind a megfelelő videokártyákra, ill. a megfelelő driverekre, fejlesztő környezetekre. Az előző féléves témalaborunk esetén megismertünk és összehasonlítottunk két lehetséges környezetet: az OpenCL-t és a CUDA-t. A munkánk során arra jutottunk, hogy habár a CUDA csak az Nvidia bizonyos GPU -in futtatható, az OpenCl-lel ellentétben, amely minden kártyán képes futni, jóval könnyebb átlátható, jobban olvasható kódot írni CUDA segítségével. (ez nyilvánvalóan adódik abból, hogy nem akarta az Nvidia, hogy minden kártyán fusson az ő környezetük és így a sajátjukra tudták optimalizálni a saját felhasználóbarátabb környezetüket tudták megvalósítani) Emellett pedig a gyártó számos debugolásra, optimalizálásra használható eszközt is ad a csomagban, ami szintén nagyban tudja segíteni a fejlesztést.

Milyen területen lehet használni a GPU-n való számításokat? Az egyik ilyen GPGPU alkalmazási terület a fizikai szimulációk. Mi a témalaborunk és idén az önlabor témánk keretein belül is ezt a területet vizsgáltuk meg, egész pontosan a folyadékszimulációt két különböző megközelítéssel: áramlás alapú és részecske alapú megközelítéssel.

Folyadékszimuláció

Napjainkban elég felkapott téma a folyadékok szimulálása (computational fluid dínamics – CFD) a számítógépes grafikában. A tudósok arra fókuszálnak, hogy minél jobb módszereket találjanak a szimulációra, megjelenítésre. A közös az összes módszerben, hogy valamiféle matematikai egyenletek segítségével szeretné leírni a folyadék mozgását. A folyadékot gyakran az egyik legösszetettebb, valósághűen a legnehezebben szimulálható jelenségnek mondják. Offline szimulációk a leggyakoribbak, amelyek lehetnek mind részecske alapú, mind rács alapú, vagy akár a kettőt ötvöző megoldások is.

Napjainkban talán a rács alapú az elterjedtebb megközelítés. Azonban ha valósidejű 3D-s szimulációt szeretnék akkor nem ez az optimális megoldás. Erre a problémára vezették be a részecskéket. Jelenlegi tudásunk szerint a részecske alapú megközelítés egy jó választás lehet, ha kisebb méretű folyadékot szeretnénk szimulálni, amelyen esetleg interakciókat is hajtunk végre. Emellett a folyadékok felszínét is részecskékkel lehet valóságosabban szimulálni.

Alapvetően a folyadékszimulációnak két része van: a szimuláció és a megjelenítés. Mivel a folyadékok megjelenítésére már nagyon sok megoldást született (pl. ray-tracing) amely élethű végeredményt hoz mi jelenleg a szimulációs lépésre fektettünk nagyobb hangsúlyt.

Most pedig lássuk részletesebben is mi rács-alapú ill részecske-alapú megközelítés:

Rács alapú vs. Részecske alapú (SPH)

**Rács alapú *(röviden)***

A témalaborunk keretén belül foglalkoztunk az ilyen módon végzett folyadékszimulációval. Ebben az esetben az alapvető gondolat az, hogy felosztjuk a folyadékunkat egy négyzetrácsra (csak 2D ban készült el végül, de a 3D megvalósítás teljes analóg) és azt határozzuk meg, hogy az adott mezőn milyen irányú és nagyságú a folyadék sebessége, amely majd szállítani fogja az anyagot. Ez az úgynevezett *Euler-i folyadék* megközelítés. Az erőssége, hogy bizonyos fizikai tulajdonságokat jól le lehet vele írni pl. nyomás, sűrűség a rácsnak köszönhetően, ugyanakkor a hátránya is éppen a rácsban rejlik, a folyadék nem képes kilépni ebből, csak arra a térfogatra van szorítva, amelyen a rácsot definiáltuk. Persze ma már mindenféle adaptív rácsokat is kitaláltak, de mi ezekkel nem foglalkoztunk a témalabor keretében.

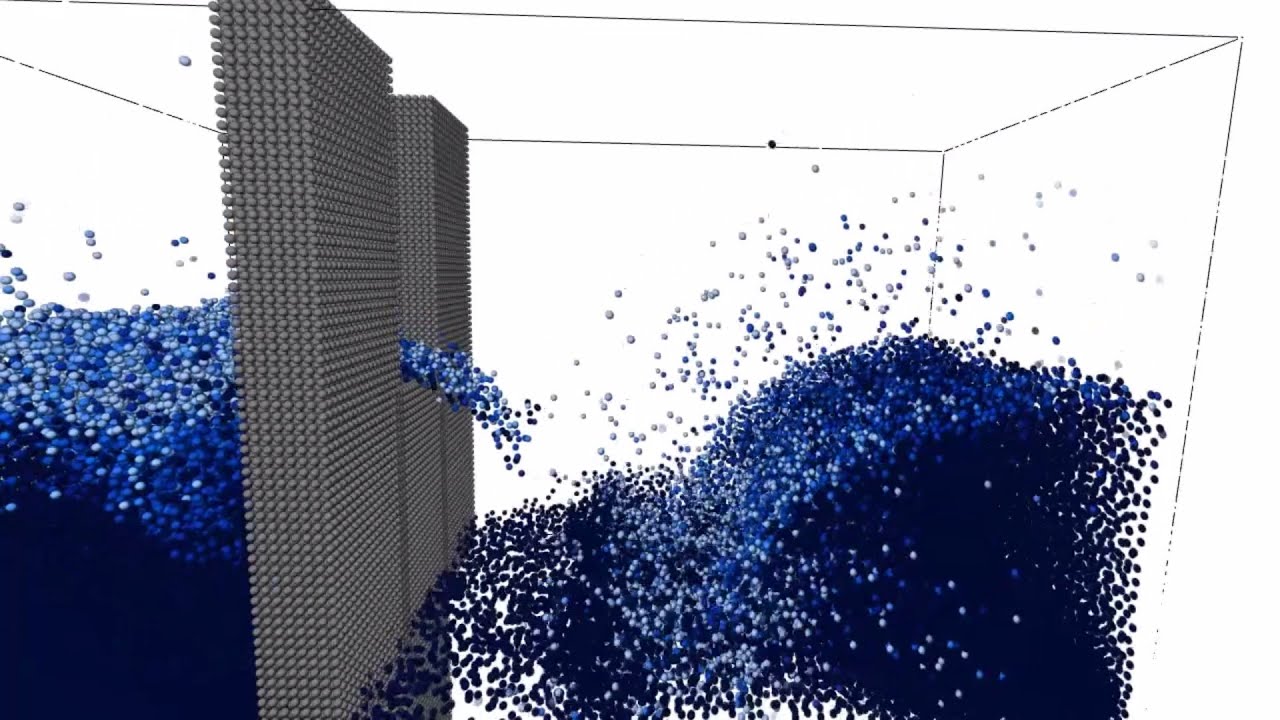
A szimulációs lépések a Navier-Stokes egyenletekre azok megoldására épülnek. Az egyenleteket Claude Navier és George Gabriel Stokes állították fel 1822-ben a folyékony anyagok mozgásának, áramlásának leírására. Alapvető elképzelésük az volt, hogy az anyagban fellépő feszültségnek két összetevője van: a folyékony anyag sebességgradiensével arányos diffúziós (vagyis egy a viszkozitást jellemző) kifejezés összetevőből és egy nyomás összetevőből áll. Mi a szimulációs lépésekben ezeket számoljuk és ez alapján jelenítjük meg a folyadékunkat.



Ennek a megközelítésnek a jellentősége abban rejlik, hogy az egyenletek mind elméleti mind gyakorlati haszonnal bírnak hiszen segítségükkel leírható pl. az időjárás, óceánokban áramlatok, repülőgépek szárnyai körül észlelt áramlás, vagy például csillagok galaxisokon belül leírt mozgása is. A repülőgépek és gépjárművek tervezése mellett használhatóak atmoszferikus szennyezés felmérésére, sőt akár véráram szimulálására is.



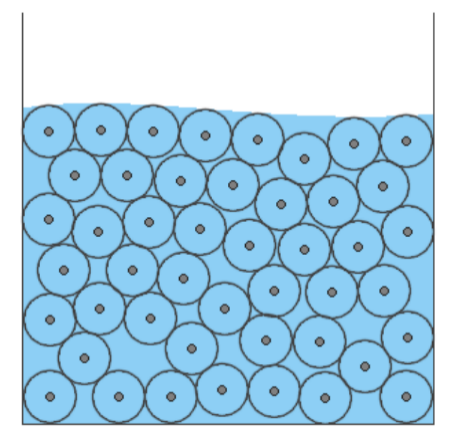
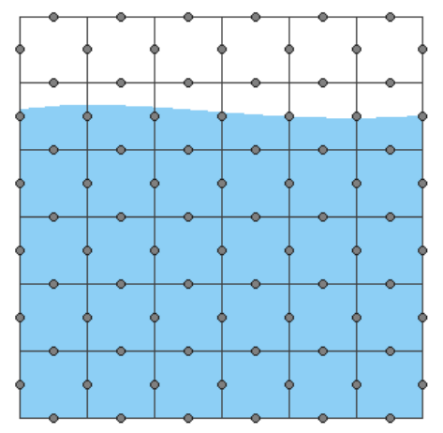
**Részecske alapú - SPH**

Az SPH (smoothed particle hydrodynamics) eredetileg az asztrofizikából származik. Széles körben használják olyan jelenségek szimulálására, amelyek így könnyebben kifejezhetővé, megérthetővé válnak. Ez egy olyan interpolációs módszer, amely a különböző értékeket és a különböző folytonos értékek deriváltjait úgy határozza meg, hogy diszkrét mintavételezési pontokat használ. Ezek a minta pontok a részecskék, amelyek számos fizikai tulajdonságot hordozhatnak pl. tömeg, pozíció, sebesség stb. De a részecskék akármilyen egyéb fizikai tulajdonságot is szállíthatnak a problémának megfelelően, mint pl. hőmérséklet, nyomás stb.

Az önlaborunk során első körben egy 2D valós-idejű szimulációt implementáltunk, amely a CPU-n fut. Majd ezt ültettük át 3D és CUDA segítségével a GPU-ra.

Lagrange-i folyadékok

A rács alapú megközelítés esetén láthattuk, hogyan épül fel az Euler-i megközelítés összenyomhatatlan folyadékokra. Ha rácsok helyett részecskéket használunk az jelentősen egyszerűsítheti az egyenleteket. A szimulációnk során azt feltételeztük, hogy az összes részecske szám állandó, ill. hogy egy részecske tömeg változatlan marad a szimuláció során ebből következik, hogy a tömegmegmaradás törvénye mindig érvényes marad. Ezzel rögtön egy egyenletet kiiktattunk az Euler-i megközelítésből ahol a rácsalapú megközelítés miatt erre is külön számolni kellett. Egy másik jelentős könnyítés abból adódik, hogy a részecskék maguk meghatározzák a folyadékot. Az Euleri nézethez képest ez azt jelenti, hogy amíg az Euleri folyadék esetén a fizikai tulajdonságok függtek az időtől és a rácson elhelyezkedő pozíciótól is ebben az esetben minden fizikai tulajdonság csak az időtől (*t*) fog függeni. (ami szintén az egyenleteket egyszerűsíti)



2. ábra Lagrange-i folyadék

1. ábra Euler-i folyadék

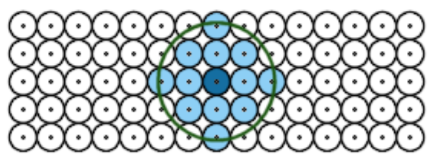
Szimuláció

Az fentiek alapján az első lépés a szimuláció CPU-n való implementálása az említett részecskealapú folyadékszimulációnak. Ez technikai szempontból azt jelentette, hogy a használt nyelvet mi választhattuk. Választásunkat elsősorban (a szakmai háttér mellett) a performancia befolyásolta a legjobban, hiszen a szimuláció egyes lépéseiben rengeteg számolás megy végbe, és adott időn belül megfelelő számú lépésnek kell történnie annak érdekében, hogy a szimuláció valós idejű legyen. Az elmondottak alapján a C++ nyelvre esett a választásunk. Szerencsére a modern C++ rengeteg olyan funkcióval rendelkezik, ami megkönnyítette az algoritmusok gyakorlatba átültetését.

Elméleti háttér és segítségek

**Szomszédos részecskék vizsgálata**

A folyadékot alkotó részecskék viselkedését legjobban a belső hatások befolyásolják, vagyis a részecskék egymás közötti kölcsönhatása. Ilyen például, hogy egy adott részecskére vonatkozó tömegsűrűség, amit a körülötte lévő részecskék befolyásolnak. Értelemszerűen minél messzebb van egy környező részecske, annál kevésbé van hatással a vizsgált részecskére. Felmerül a kérdés, hogy mi legyen az a távolság, amin kívül ez a hatás már elhanyagolható a számolás szempontjából, így növelve a teljesítményt. Ez a távolság a ***h***(angolul – *smoothing kernel support radius*), ennek a megfelelő meghatározása kulcsfontosságú, hiszen ellehetetleníti a szimulációt, ha ez a konstans túl kicsi, illetve az is ha túl nagy. Intuitívan gondolkodva a ***h*** meghatározása visszavezethető arra, hogy a vizsgált részecske körül hány részecskét vegyünk figyelembe a számolásnál. (tekintsünk *h*-ra mint a vizsgált részecske körüli gömb sugarára, ebben a gömbben kell elférnie adott számú részecskének, az elférést a részecskék száma és a térfogat segítségével tudjuk meghatározni, vagyis, hogy adott térfogatban hány részecske fér el). A problémát az **alábbi ábra** szemlélteti, a síkon.



**Téralapú hash-elés**

Miután meghatároztuk, hogy hány szomszédos részecskét akarunk átlagosan figyelembe venni, és az ehhez szükséges fő metrikát (***h***), a következő problémával állunk szemben: van valamiféle adatstruktúránk, ami tárolja a részecskéket, a később ismertetett számolások és szimulációs lépések során ezeken iterálunk végig és soknál szükség van a környező részecskék lekérdezésére. Azonban egy adott részecske és a körülötte lévő szomszédos részecskék között az adatstruktúrában semmiféle összefüggés nincs általában (az első pillanatban feltehetően van a szabályos inicializálás miatt, azonban ez egyből módosul). Ez érthető, hiszen a részecskék helye az adatstruktúrában nem áll kapcsolatban a részecskék térbeli helyével.

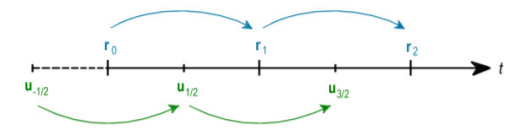
Ezen probléma megoldására szolgál a ***téralapú hash-elés***. Az alapja egy olyan hash függvény, ami a térben egymáshoz közel elhelyezkedő részecskékhez azonos hash értékeket rendel. Ez a részecskék pozícióját és a ***h***-t használja fel.

Egy ilyen hash függvény segítségével felépíthetünk egy olyan hash táblát, amit felhasználva tudunk a részecskék térbeli helyét felhasználó lekérdezéseket konstruálni. Magyarul, ha adott egy részecske, akkor a pozíciója alapján megkaphatjuk a környező részecskéket, úgy, hogy ezek átlagos száma előre meghatározott.

**Leap Frog Integrator**

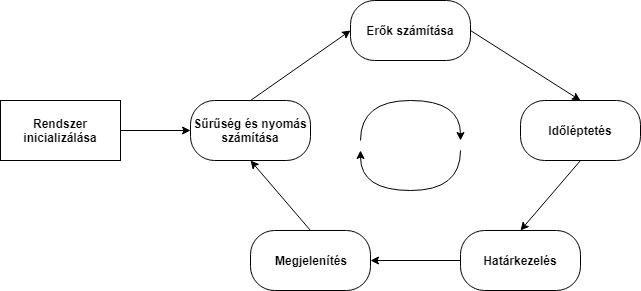
Ahhoz, hogy a folyadék a szimuláció során folyjon, az időnek telnie kell, ami azt jelenti, hogy minden szimulációs lépésben egy előre meghatározott, konstansnak vett ***Δt***-vel léptetjük az időt. Az idő léptetést felhasználva numerikus integrációval egyszerűen kiszámolható az adott részecske új sebessége és pozíciója az előző sebesség és pozíció értékéből.

A ***leap frog*** módszer onnan kapja a nevét, hogy a sebesség számolásánál használt időpillanatok „átugrnak” a pozíció számolásánál alkalmazott időpillanatokon. Ez azért van mert egy adott ***t*** időpillanat után a következő lépésben a sebességet ***t + Δt*** időpillanatban számoljuk, a pozíciót pedig ***t + Δt*** időpillanatban. Az adott ***t*** pillanatban lévő sebességet egyszerű számtani közép számolásával kaphatjuk meg. Ez a módszer pontosabb értékeket eredményez, melyek jobban megközelítik a folytonos valóságot, ennek a miértjére jelen keretek között nem térünk ki. Az **alábbi ábra** szemlélteti a sebesség és pozíció számolásához felhasznált időpillanatokat.



Áttekintés

A fentebb megismertek alapján a szimuláció középpontjában a részecskék állnak, amik számos tulajdonsággal rendelkeznek, végsősoron minden lépésben ezek fognak megfelelően a számolások alapján változni, minden lépés alapvető eleme, hogy az összes részecskére elvégezzük a számításokat (ez adja később az alapját a párhuzamosításnak, így a GPU-n való implementálásnak). Az **alábbi ábra** szemlélteti a szimuláció lépéseit a legmagasabb absztrakciós szinten. A továbbiakban ezt tekintjük át részletesebben.



Mint láthatjuk a szimuláció folyamata két nagy részre osztható, egy inicializálási részre (lépésre), továbbá egy szimulációs ciklusra, ami az inicializálás után folyamatosan ismétlődik.

Inicializálás

Az inicializálás első lépése, hogy a számolás során használt konstansokat (mind fizikai, mind egyéb) meghatározzuk. Ezek egy része adott fizikai mennyiség (pl. anyag sűrűsége, gravitációs gyorsulás stb…), vagyis eleve adott, viszont néhány konstans általunk meghatározott (a legegyszerűbb példa erre a részecskék száma), amelyeket ezáltal érdemes úgy megválasztani, hogy az a mi érdekeinket megfelelően szolgálja.

A számolások során használt konstansok meghatározása után létrehozzuk a részecskéket, melyeknek beállítjuk a tulajdonságaik kezdő értékeit. Ezek a tulajdonságok a pozíció, sebesség és a konstans tömeg, ezekből határozzuk meg a többi tulajdonság értékét (mint például a részecskék gyorsulása, vagy a rájuk ható erők).

Ebben a lépésben kap értéket a ***h*** konstans is, amelyről fentebb esett szó, és alaphelyzetbe kerül a téralapú hash-eléshez használt adatstruktúra, valamint az időléptető.

Sűrűség és nyomás kiszámítása

Ez a fizika mértékek számolásának az első lépése. Minden részecskén végigmegyünk es elvégezzük a következőket. Vesszük a környező, szomszédos részecskéket. Ezen szomszédos részecskék segítségével kiszámoljuk az adott részecske tömegsűrűségét (sűrűségét). Miután ezt kiszámoltuk, ennek és a nyugalmi sűrűségnek (konstans) a segítségével kiszámoljuk a nyomást (szintén a részecske tulajdonsága).

Erők kiszámítása

A részecskékre ható erők két nagy csoportba sorolhatók, belső és külső erők. A belső erők a nyomásból és a viszkozitásból erednek, míg a külső erők a mi esetünkben a gravitációból és a felületi feszültségből ered. Gázok esetében még ehhez hozzájönne a felhajtóerő.

A belső erők kiszámításánál újra végig iterálunk a részecskéken és mindegyik esetében megkeressük a környező részecskéket. Ezen szomszédos részecskék segítségével kiszámoljuk mind a nyomásból, mind a viszkozitásból eredő erőket, majd ezeket összegezzük.

A külső erők számítása hasonló módon történik, azzal a különbséggel, hogy a gravitációs erő nem függ a környező részecskéktől.

Az egyes részecskékre ható eredő erő a két kiszámolt komponens összegeként áll elő.

Időléptetés

Miután kiszámoltuk a megfelelő fizikai mennyiségeket az adott időpillanatban, ezt a pillanatot tovább kell léptetnünk. Ez a léptetés egy ún. ***Leap Frog Integrator*** segítségével történik (lsd. fentebb). A fizikai mennyiségekkel segítségével kiszámoljuk (továbbléptetjük) a részecskék sebességét, és pozícióját, melyek így már az új, következő időpillanatra érvényesek. Enélkül a lépés nélkül értelemszerűen minden egyhelyben állna, a szimuláció nem haladna előre az időben.

Határkezelés

A szimulációban szereplő részecskék korlátozó határok nélkül szabadon mozoghatnának a térben, ami praktikusan azt jelentené, hogy a rájuk ható gravitáció miatt a végtelenségig gyorsulnának lefelé (hiszen a szimulációban a gravitáció állandó és nincs légellenállás). Ennek nyilvánvalóan nem sok hasznát vennénk, ezért szükségünk van valamire, ami korlátozza a részecskéket a térben. Ez a gyakorlatban legegyszerűbb esetben egy tartályt jelent, amiben a folyadék mozoghat. A tartály és a részecskék interakciója a határkezelés problémája. Jellegében kissé más, mint a többi szimulációs lépés, hiszen eddig az összes lépés a fizika törvényei alapján meghatározta az anyag viselkedését, nem számolva a korlátokkal, ezzel ellentétben, most magukat a korlátokat kell bevezetnünk.

A fő cél értelemszerűen, az, hogy a folyadék, vagyis a részecskék a tárolóedényben maradjanak. Ezt úgy érhetjük el legegyszerűbben, hogy minden lépésben miután a léptetés megtörtént és a részecskék új helyre kerültek, ellenőrizzük, hogy ez belül van e a tárolóban. Ez bár elméletben egyszerűen hangzik, analitikusan elvégezve, ha nem jól leírható alakzatokkal, testekkel dolgozunk, akkor egyáltalán nem triviális. A mi esetünkben, hogy ez ne okozzon sok többlet nehézséget, így egy gömb alakú használtunk. Ennél analitikusan a gömb implicit egyenlete alapján egy pontról (a mi esetünkben az adott részecske helye a térben) egyszerűen eldönthető, hogy belül, illetve kívül helyezkedik e el.

A léptetés után kívül elhelyezkedő részecskéket vissza kell helyeznünk a tárolón belülre. Ez a visszahelyezés úgy történik, hogy egy leegyszerűsített módszerrel meghatározzuk a részecske kilépési pontját a felületen és oda helyezzük vissza. A visszahelyezett részecske sebessége értelemszerűen megváltozik. Az új sebesség megválasztásánál szimulálni szeretnénk a tartály falával történő ütközést, ehhez figyelembe kell venni az átlépés mélységét (mennyire ment ki a részecske a tartályból) és a részecske eredeti sebességét. Ezek segítségével modellezhető e megfelelő ütközés.

A gyakorlatban ez a leegyszerűsített határkezelés nagyban hozzájárul a program és a szimuláció hatékonyságához, azonban pontatlanságokkal és hibákkal jár. A mi esetünkben, a legnagyobb gondot az jelentette, hogy a tartályból kilépő, ezért a felületre visszahelyezett részecskék egy része később nem tudott a falról eltávolodni, mondhatni „hozzáragadtak” a falhoz. Ezen a probléma orvosolásának céljából, a határkezelés egy másik megközelítését is alkalmaztuk, miszerint a határt modellezzük szintén részecskékkel. A határrészecskék azonban kissé speciálisak, mert fixek. A fix határon lévő részecskék taszítva hatnak a többi átlagos részecskére, ezáltal benntartva őket a tartályban. Ez a megközelítés a szimuláció számolási pontatlansága miatt (az idő nem lehet folytonos, hiszen az időléptetést nem lehet a végtelenségig kicsinyíteni), nem tudja megfelelően biztosítani, hogy a részecskék mindig a határokon belül maradnak.

A fent elmondottak alapján végül mindkét módszert alkalmaztuk, így az előnyeik összeadódnak és megfelelő viselkedést érünk el, hiszen az első módszer biztosítja, hogy a részecskék mindig bent maradnak a határokon, míg a második megoldja a határra ragadás problémáját a befele taszítással.

Megjelenítés

Az utolsó szimulációs lépés a megjelenítés. Ekkor a kiszámolt tulajdonságokat felhasználva valamilyen módon vizualizáljuk a részecskéket, ez rengeteg féleképp történhet, bővebben a *Vizualizáció* részben írunk róla.

Összegzés

Láthatjuk, hogy a szimulációs lépésekben közös, hogy végigiterálnak a részecskéken és úgy végeznek különböző műveletet rajtuk. Soros végrehajtással processzoron, ez azt jelenti, hogy egyenesen arányos a szimuláció sebessége a részecskék számával így, ha szemelőtt tartjuk a valós időben való futást, akkor hamar korlátokba ütközünk.

Ezen megfontolások miatt lehet alkalmas számunkra a GPU-n való implementálás, hiszen míg a processzor legnagyobb erőssége a sokféle különböző bonyolultságú feladatok egymás után futtatása, a GPU ezzel szemben a sok egyszerre történő (párhuzamos) egyszerű feladatok elvégzésében jeleskedik. Egyértelműen látszik, hogy az utóbbi helyzet áll fenn. A következőekben a szimuláció GPU-ra történő implementálását tekintjük át. Ezzel a részecskék számának növelése (egy ideig, bizonyos határokat figyelembevéve) nem jelent korlátot.

Implementáció

CPU

A fentebb leírt lépések és algoritmusok implementálása C++ nyelven, párhuzamosítás nélkül, egy szálon. A fő szempont az objektumorientáltság és a C++ funkcióinak a kihasználása volt, amelyek nagyban megkönnyítették a szimuláció gyakorlatba való átültetését. Az osztálystruktúráról röviden: a ***Particle*** osztály reprezentálja, ez tárolja a részecskék szempontjából szükséges fizikai tulajdonságokat, amik: pozíció, éppeni és előző sebesség, részecskére ható erők eredője, nyomás, tömegsűrűség.

Ezeket a részecskéket menedzseli a ***Particles*** osztály, ami tárolja a részecskéket magába foglaló adatstruktúrát és hash táblát, csak, úgy mint a határt alkotó speciális részecskéket.

A szimulációt az ***SPH\_Simulator*** osztály végzi, amiben implementálva vannak a szimulációs lépések.

Az objektumorientáltságnak köszönhetően a legfontosabb elem, a „*végezzük el minden részecskére”*, meglehetősen egyszerűen véghezvihető, egyedül a tároló végig iterálásával jár, amiben a részecskék referencia szerint vannak eltárolva, így értékük egyszerűen és maradandóan változtatható. Továbbá nagy segítséget jelent, hogy a részecskék tulajdonságai objektumokba vannak zárva, így referenciákon keresztül mindig és könnyen elérhetőek.

Megvalósítás szempontjából az standard C++ tárolók jól használhatóak és levették a vállunkról a dinamikus memóriakezelés nehézségének egy részét.

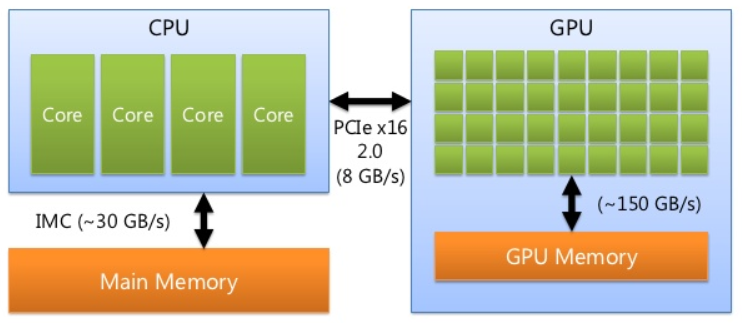
GPU

A már említett párhuzamosíthatóságok miatt kézenfekvő a GPU-ra történő implementáció a jelentős teljesítményemelkedés reményében. A GPU programozásához a már említett *NVIDIA CUDA* keretrendszert használtuk. Ez egy C alapú programozási interfészt biztosít, amiből következik, hogy nem arra tervezték, hogy objektumorientált megoldásokat alkalmazzunk rajta.

**Adatstruktúra**

A GPU programozáshoz használt adatstruktúrák egy sokszor használt formája a bufferek, vagyis tömbök. Ezeket jól és átláthatóan fel tudják használni a párhuzamos algoritmusok. Az előbbiek alapján az objektumorientált szemléletet (miszerint az egy részecskéhez tartozó adatok, tulajdonságok objektumokba zárva vannak, és úgy vannak az objektumok egybe tárolva) át kell alakítani a buffer, vagyis tömb alapú szemléletre. Ennek a szemléletnek az alapja, hogy nem a tulajdonságok halmazát csoportosítjuk, hanem az egyes tulajdonságokat magukat, vagyis minden részecske tulajdonsághoz (pozíció, sebesség, stb…) tartozik egy buffer, vagyis egy részecskék számának megfelelő méretű tömb. Ezáltal végül nagyvonalakban a tulajdonságok számának arányos bufferrel, tehát tömbbel dolgozunk. Ebben a formában egy részecskét, egy index azonosít, amely meghatározza, hogy a bufferek hányadik eleme tartozik hozzá, ezek a részecskére vonatkozó tulajdonságok.

A tömböknek szükséges memória helyet a grafikus eszköz memóriájában előre le kell foglalni, melyet speciális, a keretrendszer által nyújtott metódusokkal tehetünk meg. A foglalás után kapunk egy pointert, ami a lefoglalt terület elejére mutat. Rendkívül fontos, hogy ez a memóriacím a GPU memóriájában található, vagyis a CPU-n futó kódban nem értelmezhető, hiszen másik memóriához tartozik.



**Műveletvégzés**

Az alapelv, hogy az adatokon elvégzett egyes műveletek megfelelnek a szimuláció lépéseinek. Ezekben a lépésekben, mint említettük közös és párhuzamosítható a „végezzük el minden részecskére” rész, ezek ún. kernel függvények lesznek. A kernel függvény egy speciális, a *CUDA* keretrendszer által menedzselt függvény, amit adott példányszámban párhuzamosan tudunk futtatni a grafikus processzoron. Ez a példányszám kézenfekvően a részecskék számával megegyezik, így minden példány (ami optimális esetben egy *CUDA* szálnak (*CUDA thread)* felel meg) egyetlen részecskével kapcsolatos adott számítást végez el, ez érezhetően hatékonyabb, mint a számolások egy szálon történő, egymás utáni végrehajtása. A kernel függvények paraméterben megkapják a számoláshoz szükséges buffereket, és az adott függvénypéldány a szál (*thread*) azonosítójával tudja meghatározni, hogy éppen milyen indexű részecskével kell dolgoznia, tehát a buffereket ezzel indexelheti, így kinyerve a számoláshoz szükséges adatokat.

Természetesen az egyes szimulációs lépések egy függvényben történő megvalósítása rendkívül átláthatatlan lenne, így segédfüggvényeket alkalmazunk. Felmerül a kérdés, hogy egy kernel függvényből bármilyen másik függvény meghívható e, a válasz pedig, hogy nem. Azokat a függvényeket, amik meghívhatóak a kernel függvényekből, vagyis a grafikus eszközről, eszköz függvényeknek (*device functions*) hívjuk. Ezen metódusok egy függvény fejlécben lévő megjelenésben különböznek egy átlagos metódustól, ezáltal tudja menedzselni őket a keretrendszer. Fontos megemlíteni, hogy ezekből a függvényekből is csak hasonló függvényeket, illetve kódot lehet meghívni (ezáltal például stl tárolókat nem).

Az utolsó szimulációs lépés, a megjelenítés annyiban egészül ki, hogy mivel CPU kód segítségével történik, így a GPU-ról a pozíciót tartalmazó bufferből vissza kell másolni az adatokat, hogy használhassuk, hiszen a külön memóriában közvetlen módon nem érhetjük el. A másolást a keretrendszer végzi, a megfelelő metódus meghívásával.

**Nehézségek**

A legfőbb nehézség abból fakad, hogy a GPU egy másik eszköz, máshogy lehet programozni, mind keretrendszerileg (C-hez jobban hasonló módon), mind szemantikailag (nagyfokú párhuzamosság). Mivel nem használhatunk stl tárolókat, néhány algoritmus implementálása jóval nehezebbé válik. A legfőbb problémát a téralapú hash-elés okozta, hiszen a hash tábla felépítése dinamikusan nyújtható tömböket implikál. Túllendülve a plusz munka tényén, természetesen C-ben is implementálhatóak hasonló tömbök, dinamikus memóriakezeléssel. Ez azonban a grafikus eszközön és annak memóriáján közel sem triviális, és egyáltalán nem hatékony.

A megoldást végül kompromisszumkötéssel találtuk meg, vagyis tudván hogy egy részecskének a ***h*** alapján átlagosan adott számú szomszédja van (nagyjából 30 és 50 között), így fix 100 méretű tömböket alkalmaztunk. Valószínűleg előfordul olyan, hogy több mint 100 környező részecske lenne, azonban ez a szimuláció pontosságának szempontjából elhanyagolható.

A nagyfokú párhuzamosságból egy másik probléma eredt, miszerint előfordulhat olyan eset, hogy egy buffer adott indexét egyszerre éri el több szál, és lehet, hogy valamelyik elérés írás. Ez már számtalan jól ismert problémához vezet. Szerencsére erre a megoldást jelent a régóta használatos dupla bufferezés. Ez azt jelenti, hogy minden ilyen problémával szembenálló bufferből kettő van, egy csak olvasható, és egy csak írható. Ezek segítségével a műveletvégzésekhez szükséges adatokat az olvasható bufferből nyerjük, amik nem változnak a műveletvégzés során, és az eredmény az írható (de az adott számolásban nem olvasott) bufferbe kerül. Természetesen a művelet után a két buffernek szerepet kell cserélnie, hiszen a számolási eredményeket tartalmazó buffer lesz az érvényes adat szempontjából, így a következő lépésnek ebből kell kinyernie a szükséges értékeket.

**Optimalizálás**

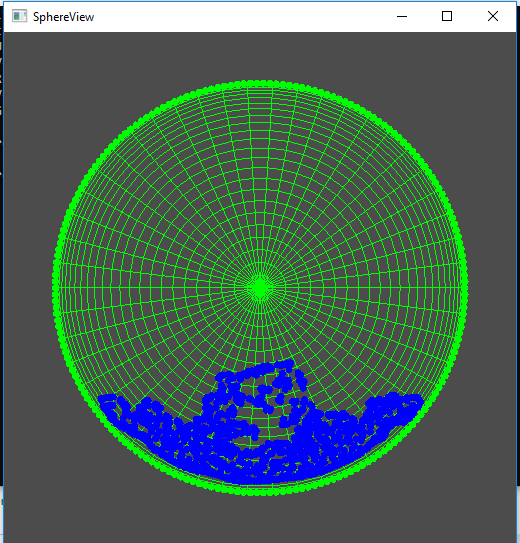
Mivel az implementáció elsőként CPU-ra történt, így tartalmaz olyan lépéseket és algoritmusokat, amelyek nem teljesen optimálisak párhuzamos környezetben. A legszembetűnőbb ilyen algoritmus a téralapú hash-elés.

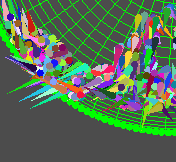
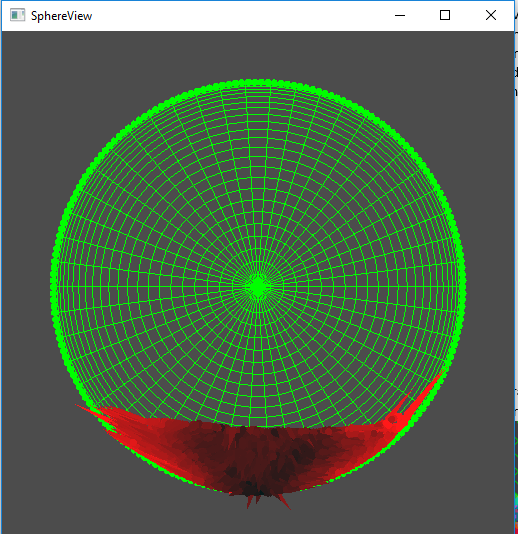
Ennek során ugyanis végigmegyünk a részecskék pozícióján, és a részecskék azonosítóját a téralapú hash függvény segítségével elhelyezzük a hash táblába. Az egész lényege ugye, hogy a hash tábla egyes helyei listák, ahol az azonos hash értékű részecskék azonosítói elhelyezkednek. Azonban ezen lista feltöltése párhuzamos környezetben nem triviális. Nem adhatjuk párhuzamosan hozzá a megfelelő listákhoz az egyes azonosítókat, hiszen, mi történik, ha egy adott listába egyszerre több szál szeretne beszúrni. Emiatt kezdetben ez a lépés sorosan működött, csak úgy, mint a CPU-n, ez azonban érezhetően hatékonyságbeli visszaeséssel járt.

Szerencsére a keretrendszer szolgál olyan megoldásokkal, amivel az algoritmus párhuzamosítható. Nekünk az lenne a fontos, hogy amíg az egyik szál beszúrja a részecskét, addig másik szál ne szúrhassa be, ehhez ***atomi*** ***műveleteket*** kell használnunk, amelyeket a *CUDA* bizonyos formában támogat.

Vizualizáció

Habár az önlaborunk során a legnagyobb hangsúlyt a szimulációra fektettük érdemes pár szót ejteni a megjelenítésről is. A megjelenítéshez a projekt során az OpenGL által nyújtotta lehetőségeket használtuk. A megjelenítés esetén két fő kérdés merült fel: honnan éri el az OpenGL a részecske pozíciókat ill. milyen alakzattal, milyen módon rajzolja ki azokat.

A 2D-s CPUn implementált verzió esetén a részecskék pozíciója a számítógép memóriájából könnyedén elérhetőek voltak hiszen a CPUn való szimulációs lépések itt dolgoztak rajtuk. A számítógépes grafikában a megjelenítés számos részből áll gondolok itt shaderek írásásra, bufferek foglalására kiosztására. Mivel mi a lehető legegyszerűbb megjelenítést szerettük volna, kihasználtuk, hogy az OpenGL nyújt néhány alapvető funkciót amivel ezeket a műveleteket „megúszhatjuk”. Miután a memóriából kinyertük a pozíciókat egy egyszerű *glBegin(GL\_TRIANGLE\_FAN);* parancs után a megfelelő koordinátákat felsorolva megkaptuk a részecskéket reprezentáló kis köröket. Az eredmény a következően nézett ki:

A debugolás során arra jutottunk hogy szükséges lenne valamilyen módon a részecskék sebességét is vizualizálni, ezért a következő megoldások születtek:

A 3D-s megjelenítés során a szimuláció már a GPU-n futott ennek következtében a részecske pozíciók is a GPU memóriában voltak elérhetőek. Ezeket először a számítógép memóriába másoljuk majd úgy adjuk az OpenGL-nek. Ez látható, hogy felesleges lépés hiszen a GPU memóriából a számítógép memóriába majd vissza a GPU memóriaba másoljuk az adatot, azonban a jelenlegi részecskeszámmal és szimulációval nem jelent jelentős plusz időt. A megjelenítéshez az OpenGL beípétett függvényiet használjuk:

* *glutWireSphere(0.8, 50, 50);* - ami egy ráccsal kirajzolt gömb
* *glTranslatef(r.x, r.y, r.z);* - a gömbök eltolásához
* Ill. egyéb a kamerakezelést szolgáló beépített függvények (*gluPerspective*, *gluLookAt*).

Az eredményként kapott megjelenítés: (amelyben a WASD gombokkal a kamerát mozgathatjuk):

